REMARKS/ARGUMENTS

These remarks are made in response to the Office Action of September 21, 2004 (Office Action). As this response is timely filed within the three-month statutory period, no fee is believed due.

In paragraph 1, claims 4 and 5 were rejected for minor informalities that have been corrected. Applications respectfully request the objections be withdrawn.

In paragraphs 2-3, claims 1-25 under 35 U.S.C. § 102(e) have been rejected as being anticipated by U.S. Patent No. 6,584,587 to McDermott, *et al.* (McDermott).

In response to the Office Action, Applicants have amended claim 13 to clarify that the time out susceptible tasks each execute within a designated thread of execution. Support for this amendment can be found in claims 1 and 14, and throughout the Applications' specification. Consequently, no new matter has been added.

It may be helpful to briefly review the features of Applicants' invention before addressing the rejections on the art. The Applicants' claimed invention provides a thread management solution that permits threads executing tasks to time-out under established conditions. In particular, the invention can initiate a timer corresponding to an identified task executing within a thread. The timer can execute within a separate thread. When the timer expires, a previously established time-out recovery action can be responsively executed.

McDermott's invention is directed to a hardware-based watchdog system that prevents a complex electronic system from freezing or "locking up" when a subsystem of the complex electronic system fails. McDermott contemplates that each subsystem can have a reset timer constantly running (a reset task). When the timer reaches zero, a hardware reset occurs. Each reset task can be periodically pinged from a separately executing watchdog process. When a response to the ping is returned, the watchdog process can assume the hardware associated with the reset task is not "locked up" and can reset the corresponding reset timer, so that the hardware reset is delayed. When no ping

is returned, the reset timer continues to count down, resulting in the hardware reset occurrence.

As noted at page 1, lines 23 to 25, conventional solutions for monitoring system processes have fallen short with respect to managing multi-threaded computer programs. That is, although conventional management systems can determine which process experienced an error (as stated at page 1, lines 27-30), such systems offer little insight as to which thread of a larger process is responsible for causing an error condition in a computer program.

The Applicants' claimed invention solves the problem of detecting and responding to error conditions or time-out conditions at the thread level. The level of granularity at which the Applicants' claimed invention operates is significant, as conventional monitoring systems failed to address error conditions at that level and even failed to recognize the need to operate at that level of granularity.

Like other known solutions for monitoring systems, McDermott is completely silent in regard to thread-level management or even problems making thread-level management beneficial. The term "thread" fails to appear in McDermott in any fashion. The context in which "task" is used by McDermott can be shown at column 2, lines 13-16, where a "task module" is broadly defined as a piece of software for controlling and/or monitoring a device or subsystem (e.g., a piece of hardware), or for controlling and/or monitoring other software modules. It is evident from this definition of task module, that McDermott defines task broadly as a basic unit of machine-readable code that causes a machine to perform designated programmatic actions.

Hence, in the context of McDermott the term task is synonymous with the term process. Moreover, McDermott is not even directed to managing complex computer programs, but is instead directed towards managing audio/video (home theatre) hardware components (as noted from column 2, line 17 to column 3, line 28), which is even a separate problem space from that of the present invention.

Applicants have included definitions for "thread", "task", and "process" extracted from www.whatis.com (an IT specific encyclopedia) on December 21, 2004 to emphasize the differences between the three terms, as generally used in computer science, the field of the present invention. Applicants have also included the definition of thread returned from using the GOOGLE define function (define: thread) on December 21, 2004 – the definition was referenced from www.newtolinux.org.uk/glossary.shtml so as to provide additional support the claimed term "thread".

For the remainder of this response, the terms task, thread, and process will be used in accordance with these presented definitions, which are consistent with how the terms are used in the Applicants disclosure and in McDermott.

(1) PROCESS:

A process is an instance of a program running in a computer. It is close in meaning to task, a term used in some operating systems. In Unix and some other operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program). Like a task, a process is a running program with which a particular set of data is associated so that the process can be kept track of. An application that is being shared by multiple users will generally have one process at some stage of execution for each user.

(2) TASK

In computer programming, a task is a basic unit of programming that an operating system controls. Depending on how the operating system defines a task in its design, this unit of programming may be an entire program or each successive invocation of a program. Since one program may make requests of other utility programs, the utility programs may also

be considered tasks (or subtasks). All of today's widely-used operating systems support multitasking, which allows multiple tasks to run concurrently, taking turns using the resources of the computer.

(3) THREAD (from whatis.com)

In computer programming, a thread is placeholder information associated with a single use of a program that can handle multiple concurrent users. From the program's point-of-view, a thread is the information needed to serve one individual user or a particular service request. If multiple users are using the program or concurrent requests from other programs occur, a thread is created and maintained for each of them. The thread allows a program to know which user is being served as the program alternately gets re-entered on behalf of different users. (One way thread information is kept by storing it in a special data area and putting the address of that data area in a register. The operating system always saves the contents of the register when the program is interrupted and restores it when it gives the program control again.)

(3a) THREAD (from www.newtolinux.org.uk/glossary.shtml)

A small piece of programming that acts as an independent subset of a larger program, also called a "process". A multithreaded program can run much faster than a monolithic, or single-threaded, program because several, or even many, different tasks can be performed concurrently, rather than serially (sequentially). Also, threads within a single application can share resources and pass data back and forth between themselves.

Now that an appropriate overview for the present invention has been established, and that terms as used in the present invention and in McDermott have been established, Applicants proceed to respond to specific rejections to claims 1 and 14, that claim the steps of:

> receiving a request from at least one of a plurality of client computer programs to begin a timer corresponding to an identified task **executing within a particular thread** of execution of said **client computer program**, wherein said identified task has been identified as a time-out susceptible task;
>
> starting a timer in **another separate thread** of execution which corresponds to said request and said time-out susceptible task;
>
> timing said time-out susceptible task; and
>
> if said timer expires, performing a recovery action corresponding to said time-out susceptible task.

McDermott is silent concerning threads and instead discusses task management, which, as defined above, is not the same. (Applicants again direct the Examiner to page 1, lines 23-30 of the background that states conventional monitoring systems focus upon process management, but fall short with respect to managing multi-threaded computer programs).

Accordingly, McDermott fails to disclose the limitation of beginning a timer corresponding to an identified task **executing within a particular thread** that appears in the receiving step. Column 4, lines 22-38 is referenced for this teaching, which discusses that a watchdog module 190 communicates with task modules 192, but provides no teachings regarding threads, no such teachings are explicitly or inherently present in McDermott.

Further, the Applicants claim that a timer is started (after receiving the request due to antecedent basis) within **another separate thread** of execution. The Examiner has interpreted (without support from McDermott, which is silent in regard to threads) that the watchdog module executes on a different thread than the task module. While this may be possible, many configurations are possible, and technical considerations make it UNLIKELY that McDermott would be implemented within a thread-sensitive computing environment.

Specifically, McDermott fails to expressly state the claimed limitation regarding executing a timer via a separate thread, which is not inherent in McDermott (which most likely operates in a task-oriented environment and not in a multi-threaded environment, which would be a necessary precursor for the Examiner's interpretation to be possible). Notably, typically only complex operating systems functioning in a tightly coupled hardware environment (like within a single server that supports multiple processors) use multithreaded technologies. Loosely coupled complex electronic systems (like connected audio/video components described in McDermott) would use a process based operating system (like an embedded operating system not having multi-threading capabilities) or a proprietary operating system, which again would not benefit from the complexities of a multi-threaded operating system.

Further, McDermott fails to teach that a timer is started RESPONSIVE to receiving a request – i.e. the task is originated to respond to the request. McDermott's teachings are completely different. Instead, McDermott teaches that reset timers are to be constantly running within hardware components to ensure the hardware component does not freeze up. Timers and timing tasks of McDermott do not, therefore, correspond to specific request, as claimed by the Applicants.

In light of the above, the 35 U.S.C. § 102(e) rejections to claims 1 and 14 should be withdrawn, since under 35 U.S.C. § 102, each and every claimed limitation must be explicitly or implicitly taught by the cited art. McDermott fails to explicitly or implicitly

teach starting a timer responsive to receiving a request. McDermott also fails to explicitly or implicitly teach that the timer corresponds to an identified task executing within a particular thread of execution of a client program. McDermott further fails to explicitly or implicitly teach the timer is to be started in a separate thread of execution which corresponds to the request (in fact, there is no request for the timer of McDermott to correspond to). Since independent claims 9, 13, and 22 also include the limitations of starting timers responsive to requests that correspond to tasks executing within threads, the 35 U.S.C. § 102(e) rejections to claims 9, 13, and 22 should also be withdrawn. All other claims 2-8, 10-12, 15-21 are dependent on claims 1, 9, 13, 14, and/or 22. Consequently, the rejections to claims 1-25 should be withdrawn, which action is respectfully requested.

Although the dependent claims should now be in an allowable state, Applicants take a few minutes to further differentiate dependent claim limitations from the referenced art.

Referring to claims 3, 5, 16, and 18, Applicants claim destroying a thread of execution as part of the recovery action. Destroying a "stuck" thread is not the same as destroying the process to which the thread relates. Other threads (ones that are not stuck) performing actions for the parent process would not be destroyed. McDermott describes performing a hardware reset, which is different from destroying a thread (no parent process or threads relating to the parent are able to function unabated during a hardware reset, as they can then the thread was destroyed).

Referring to claims 4, 6, 11, 17, 19, and 24 the Applicants claim restarting the time-out sensitive task. In contrast, McDermott does not restart a particular task (that was "stuck"), but instead performs a hardware reset. In McDermott any pending "tasks" would not be restarted, and might not even exist within a non-cleared < a hardware reset will likely reset RAM > memory space to be restarted.

Referring to claims 7 and 20, Applicants claim that the recovery action forces a client computer program to discontinue execution. McDermott halts execution based on a hardware reset, but does not necessarily prevent the halted device from continuing to execute (if execution specifics were queued or streamed to the reset device from a non-reset device, for example, the hardware reset would not discontinue execution, as claimed by the Applicants).

Referring to claims 8 and 21, Applicants receive a request from the computer program to stop the timer. McDermott does not teach or suggest such a request. The "push back" timer chip of column 4, lines 23-38 resets the timer, but doesn't STOP the timer, as claimed by the Applicants.

In light of the above, Applicants believe that this application is now in full condition for allowance, which action is respectfully requested. Applicants request that the Examiner call the undersigned if clarification is needed on any matter within this Amendment, or if the Examiner believes a telephone interview would expedite the prosecution of the subject application to completion.

Respectfully submitted,

Date: 12-21-04

Gregory A. Nelson, Registration No. 30,577
Richard A. Hinson, Registration No. 47,652
Brian K. Buchheit, Registration No. 50,667
AKERMAN SENTERFITT
Customer No. 40987
Post Office Box 3188
West Palm Beach, FL 33402-3188
Telephone: (561) 653-5000